Displaying Data: Overview of the “VCS” module

History of the VCS module

- A long long time ago, PCMDI developed the Visualization Control System (VCS), a motif-based GUI to read, display and perform some computation on climate model output.
- PCMDI’s mission broadened and we began creating the Climate Data Diagnosis Tools (CDAT) to further enable the climate community to manipulate climate data.
- Naturally VCS was embedded into CDAT as the “natural” module to display Numerical arrays.
- Only 1D or 2D can be represented via VCS, and it is mostly designed for 2D graphics.

VCS Concepts and Terminology

- In order to use the VCS properly and efficiently module it is important to understand its basic concepts:
- Data and other graphic object are drawn on a "**VCS Canvas**".
- The way data are represented is controlled via "**Graphic Methods**". For example "boxfill", "isofill", "isoline", "vector", etc…
- The location where things are drawn (data, legend, title, comments, units, etc..) is determined via "**Templates**"
- Additional elements can be drawn using "**Primitives**" (lines, polygons, text, markers, etc…)

VCS Concepts and Terminology

- VCS canvas needs to be initialized (created)

  x=vcs.init() # without any arguments
- You can manage create up 8 Canvas at once, when not needed a canvas should be closed:

  x.close() # without any arguments, nothing returned
- Think of Canvas as “magic board” once you finished your plot you can simply “clean” (or erase) it, there is no need to create a new canvas

  x.clear() # without any arguments, nothing returned

VCS Concepts and Terminology

- Via templates (see later) you can have multiple plots on a single Canvas.
- By default data will be plotted on the “default” template using the “default” “boxfill” graphic method (for 2d data) or the “default” “yxvsx” graphic method (for 1d data)

```
x.plot(data)
# data is usually a transient variable, but could be a Numeric
    array, MA or even a python list.
```

Setting up VCS

- VCS (and VCDAT) settings
  are stored under your $HOME directory under in the PCMDI_GRAPHIC directory. Important files are:
  - initial.attributes : It contains all predefined template and graphic methods, you can add your own…
  - HARD_COPY # printers definitions file
  - vcdat_initial.py # your vcdat default settings
  - vcdat_teaching_script_file.py and vcdat_recording_script_file.py VCDAT session files to review your last work in VCDAT

- If not present these files will be created as needed

VCS First Help (1)

- Basic help on VCS can be obtain inline via: **vcs.help()**
- This will list available function in vcs, notably functions to create/get/query vcs objects, extract of the help:

--- VCS Canvas Functions ---

Plotting: plot boxfill continents isofill isoline meshfill
outfill outline scatter vector xvsy
xyvsy yxvsx

Querying: isboxfill iscolormap iscontinents isfillarea isgraphicsmethod
isisofill isisoline isline ismarker ismeshfill isoutfill
isoutline isplot isprojection isscatter issecondaryobject
istemplate istextcombined istextorientation istexttable isvector
isxvsy isplot islandscape isportrait

Creating: createboxfill createcolormap createcontinents createfillarea
createisofill createisoline createline createmeshfill
createmarker createoutfill createoutline createprojection
createscatter createtemplate createtextcombined
createtextorientation createtexttable createvector createxvsy
createxyvsy createyxvsx

VCS First Help (2)

Getting: getboxfill getcolorcell getcolormap getcolormapname getcolors getcontinents getcontinentstype getfillarea getisofill getisoline getline getmeshfill getmarker getoutfill getoutline getplot getprojection getscatter gettemplate gettextcombined gettextorientation gettexttable getvector getxvsy getxyvsy getyxvsx

Removing: removeobject

Scripting: scriptobject scriptrun scriptstate saveinitialfile

Operating: animate cgm clear close colormapgui eps flush geometry gif graphicsmethodtype gs help init landscape listelements mkevenlevels mklabels mkscale mode objecthelp open orientation page pause portrait postscript printer pstogif raster set setcolorcell setcolormap setcontinentstype show update vcsError

VCS First Help

This also can be used to get help on a specific command:

```
vcs.help('createboxfill')
```

Function: createboxfill # Construct a new boxfill graphics method

Description of Function:
Create a new boxfill graphics method given the the name and the existing boxfill graphics method to copy the attributes from. If no existing boxfill graphics method name is given, then the default boxfill graphics method will be used as the graphics method to which the attributes will be copied from.

If the name provided already exists, then a error will be returned. Graphics method names must be unique.

Graphic Methods Concepts (1)

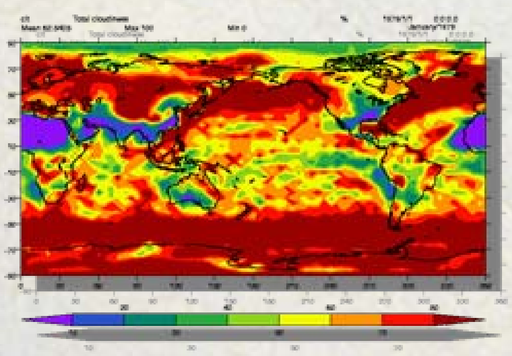
- Essentially a graphic method represents **<u>HOW</u>** data are going to be plotted (the WHERE will be determined via templates, see later)
- There are 12 type of graphic methods in VCS:
  - 2D Graphic Methods
    - Boxfill, Isofill, Isoline, Meshfill, Vector, Outfill, Outline, Continents , (Taylordiagrams ?)
  - 1D Graphic Methods
    - Yxvsx, Xyvsy, XvsY, Scatter
- A graphic method object is created from a canvas object using one of the “create” functions, for example for a boxfill:
  - gm = x.**create***boxfill*(‘name’)

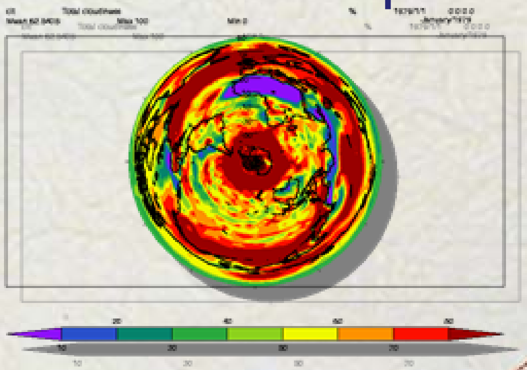
Graphic Methods Concepts (2)

- “boxfill” should be replaced appropriately with one of the 12 valid types, the ‘name’ must be unique to this type (remember the initial.attribute contains your predefined vcs objects).
- If a graphic method already exists, an error will be generated if you try to “re”create it, but it can be retrieved using one of the “get” functions:
  - gm = x.**get***boxfill*(‘name’)
- Again boxfill should be replaced with appropriate name
- gm can then be used to plot our data:
  - x.plot(data,gm)

Graphic Methods Concepts (3)

- Although there are **12 types of graphic method**, each graphic method object has a set of attributes that will control it (for example isoline graphic methods have a level attribute which controls which isocontours to draw). Therefore you can have multiple vcs graphic method object of the same type, but all representing data differently. An isoline graphic method with a projection attribute set to “default” (normal lat/lon plot) will be drastically different from one with this attribute set to “polar”





Graphic Methods Concepts (4)

- Graphic method object attributes can be listed using their own list command. All vcs object have a list command
  - gm.list()
- The following attributes are common to all graphic methods object:
  - datawc_x1 : **data 1**st **w**orld **c**oordinate on **X** axis
  - datawc_x2 : **data 2**nd **w**orld **c**oordinate on **X** axis
  - datawc_y1 : **data 1**st **w**orld **c**oordinate on **Y** axis
  - datawc_y2 : **data 2**nd **w**orld **c**oordinate on **Y** axis
  - **x**ticlabels**1**: labels to use for **first** set of labels on **X** axis
  - Same for **2nd** set and **Y** axis (e.g **y**ticlabels**2**)
  - xmtics1: location to use for first set of intermediate tic marks on X axis
  - Same for 2nd set and y axis (e.g ymtics2)
- datawc attributes take floats as values
- “tic” attributes take dictionaries as attribute (or ‘*’ for automatic)

2D - “boxfill”

- The boxfill graphic method takes a 2D array and represents it by filling a “box” (determined by the bounds on each axis values) with a color linked to the array’s value at this location:

```
b=x.createboxfill(‘new’)
x.plot(data,b)
```

2D - “boxfill”, attributes

- b.list()

Generic Info	----------Boxfill (Gfb) member (attribute) listings ----------
	Canvas Mode = 1
	graphics method = Gfb # indicates the graphic method type: Graphic Filled Boxes (Gfb)
	name = new # Name of the specific graphic method

Projection	projection = linear # projection to use (see projection section)
Labels and Ticks	xticlabels1 = * # 1st set of tic labels, ‘*’ means ‘automatic’ xticlabels2 = * # 2nd set of labels (pos determined by template) xmtics1 = # 1st set of sub ti for details) xmtics2 = yticlabels1 = * yticlabels2 = * ymtics1 = ymtics2 =
World Coordinates	datawc_x1 = 1.00000002004e+20 # world coordinate of 1st x in data area , 1.E20 means auto datawc_y1 = 1.00000002004e+20 datawc_x2 = 1.00000002004e+20 datawc_y2 = 1.00000002004e+20
Axes transformation	xaxisconvert = linear # Possible conversion of X axis , linear, log, area weighted yaxisconvert = linear # same for Y
Levels/Color association	boxfill_type = linear # Type of boxfill method can be linear, log10 or custom (see later)
Levels	level_1 = 1.00000002004e+20 # First Numeric value in the array to represent (for linar and log10 type) level_2 = 1.00000002004e+20 # second one levels = ([1.0000000200408773e+20, 1.0000000200408773e+20],) # Levels of num values to repr in custom mode
Colors	color_1 = 16 # first color to use (for lin and log10) color_2 = 239 # last color to use fillareacolors = None # Colors to associate with aach levels section
Legend	legend = None # Dictionary of values/text pair to put on the legend bar ext_1 = n # draw extension arow before fisr value/segm ext_2 = n # extension arrow after last num value/segment
Missing Values	missing = 241 # color to use for missing values

2D - create vs get

- Let's a create a new VCS boxfill graphic method
- b = x.createboxfill('new')
- The new graphic method object has been created using the "default" values, if you'd like to use an already existing boxfill graphic method as a starting point for your new boxfill method, you can pass its name when creating it:
- b2 = x.createboxfill('new2','new')

WARNING:

b2 = x.getboxfill('new') is different, modification to b2 will also modify b or any other instance of "new". Therefore it is NOT recommended to have 2 python instances of the same vcs object, and it should be avoided at all cost. The get functions should only be used to retrieve an object from the vcs memory if NOT already in Python.

2D - Preliminaries

- The following will be assumed to be typed before each section

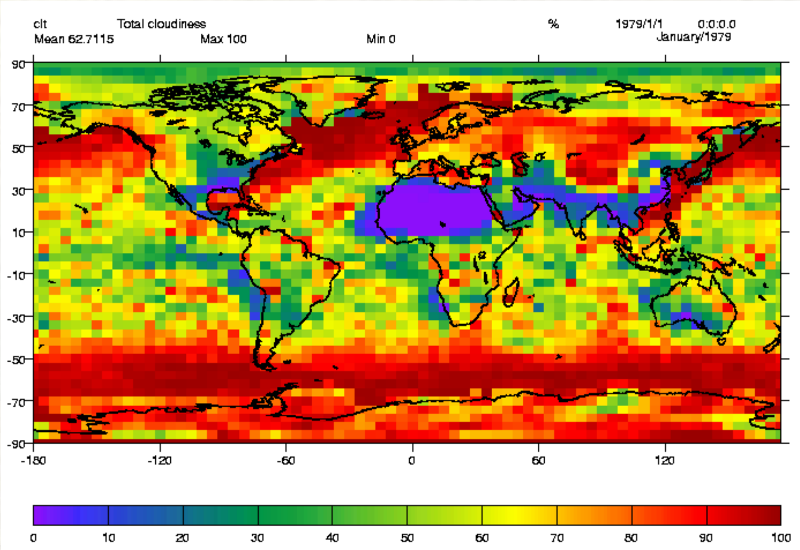
```
import sys, cdms,vcs
f=cdms.open(sys.prefix+'/sample_data/clt.nc')
data=f('clt') # 3D data
b=x.createboxfill('new')
```

- Now note that although data is 3D it can still be passed to the plot function. VCS will use the first index of every leading dimension until it gets a 2D array. Therefore:

```
x.plot(data,b)
```

  – Is equivalent to:

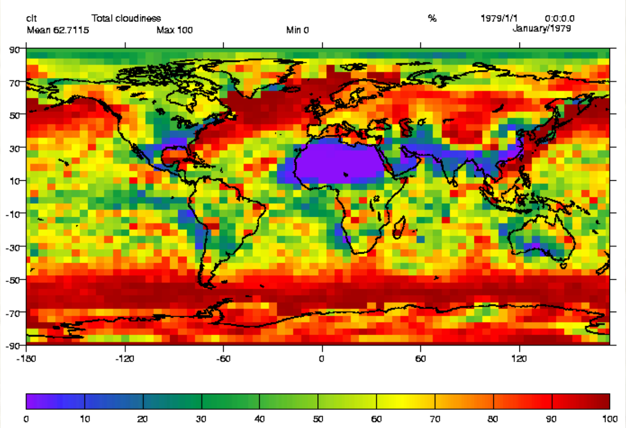
```
x.plot(data[0],b)
```

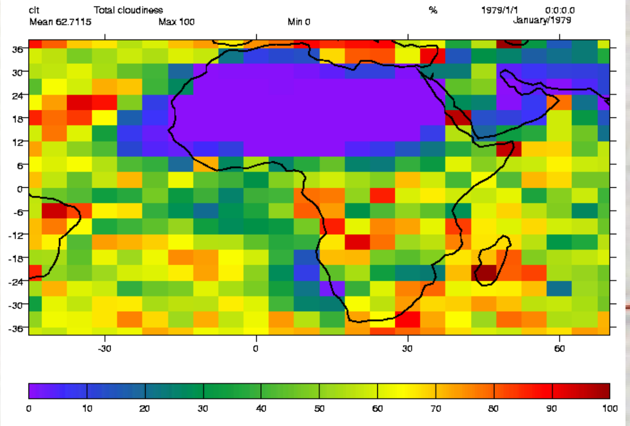


2D - World Coordinates

- The worldcoordinate attributes are present on all graphic methods, our “data” sample is global now let’s assume we’re interested in a subset area, instead of reading the data over a subset we can simply tell vcs to represent data only over this area, for example to visualize Africa:

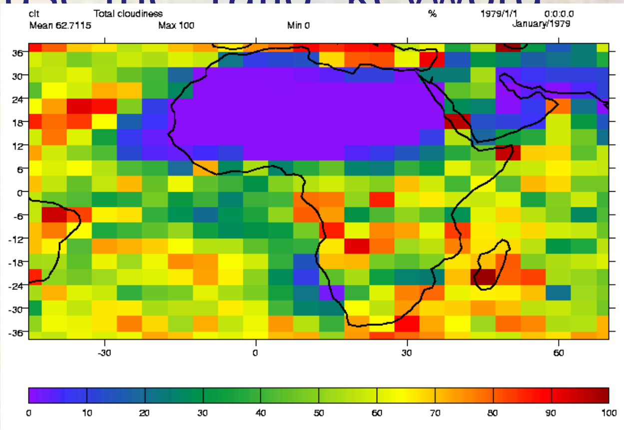
```
b.datawc_x1 = -45.
b.datawc_x2 = 70.
b.datawc_y1 = -38.
b.datawc_y2 = 38.

x.plot(s,b)
```





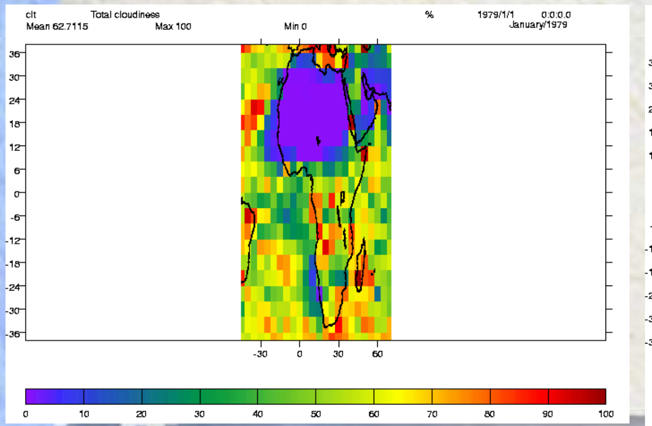
2D - Controlling ratio

- Note that altering the worldcoordinate may lead to distorted aspect ratio, we’ll see later that this can be corrected by acting on the “template”, but an easier way to do this is to use the “ratio” keyword

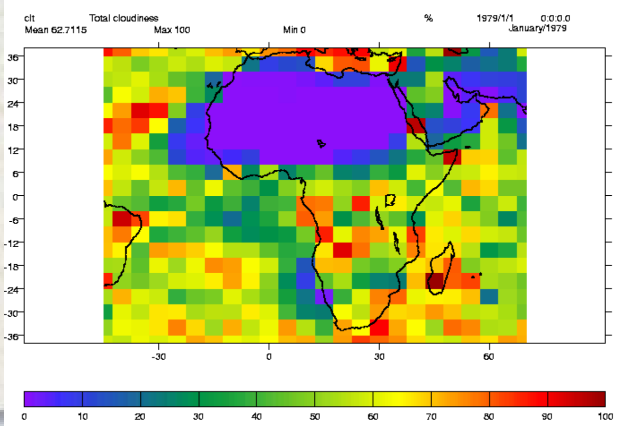


- “ratio” controls the Y ratio relative to X
- ratio=2 means Y will be twice X
- Now, for data with spatial grids, the ‘auto’ value can be passed.
- If you also want to move the box and tick marks simply add ‘t’ at the end

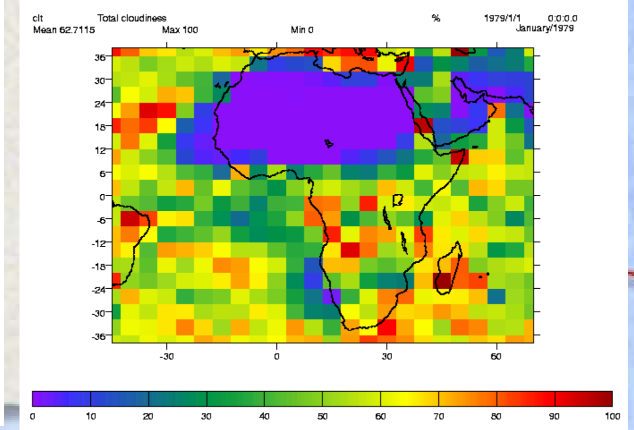
x.plot(data,b,ratio=2)



x.plot(data,b,ratio=‘auto’)



x.plot(data,b,ratio=‘autot’)

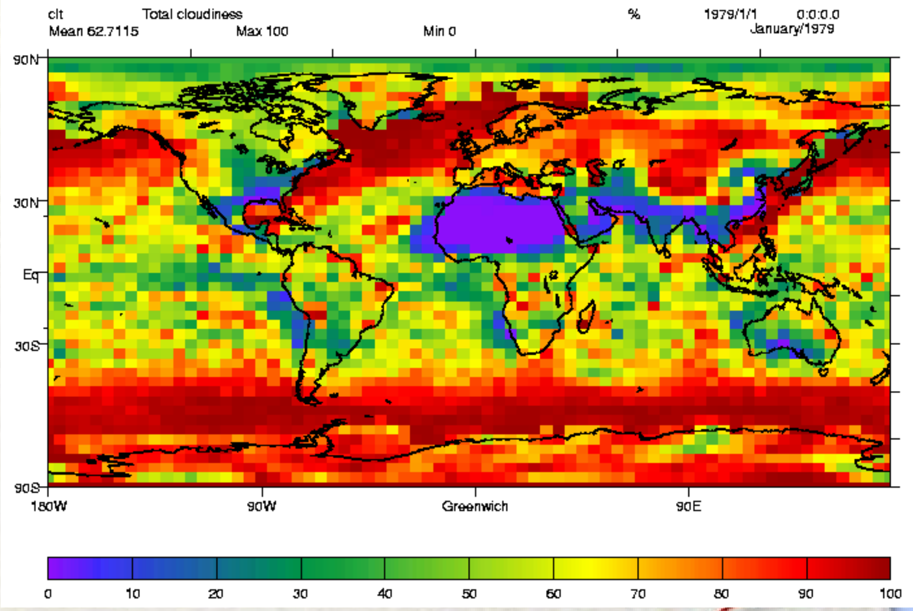


2D - Controlling Labels and Tick Marks

- Labels can be controlled via dictionaries of location/text
- For example

```
Lons={ -180:'180W', -90:'90W',0,'Greenwich',90:'90E',180:'180E'}
Lats1={-90:'90S',-30:'30S',0:'Eq',30:'30N',90:'90N'}
Lats2={-23.5:'Tropic of Capricorn',23.5:'Tropic of Cancer'}
b.xticlabels1=Lons
b.yticlabels1=Lats1
b.ymtics1=Lats2
x.plot(data,b)
```

• Note that although there is text associated with them, no labels are written next to the 'Tropics', since they have been associated to the sub tick marks (ymtics1), and also that the length of these ticks is different from the length of the main tics (yticlabels1)



2D - Axes Transformation (1)

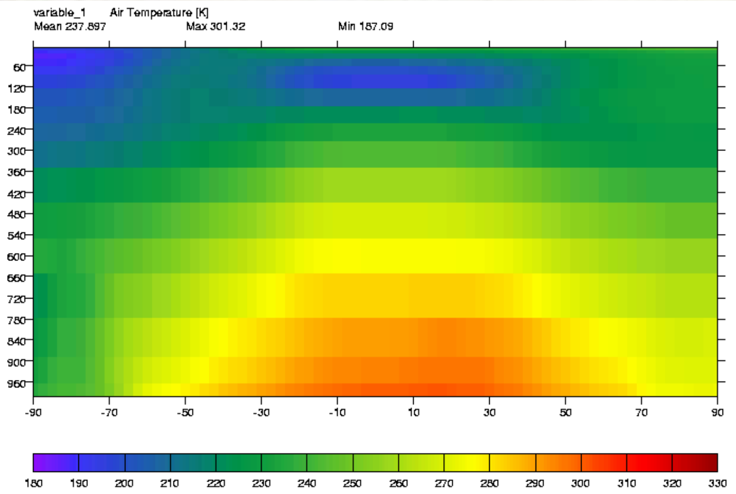
- Sometimes you need to apply a transformation on axes, vcs allows you to apply an “area-weighted”,”ln”, “log10”, or “exp” transformation, on the fly.

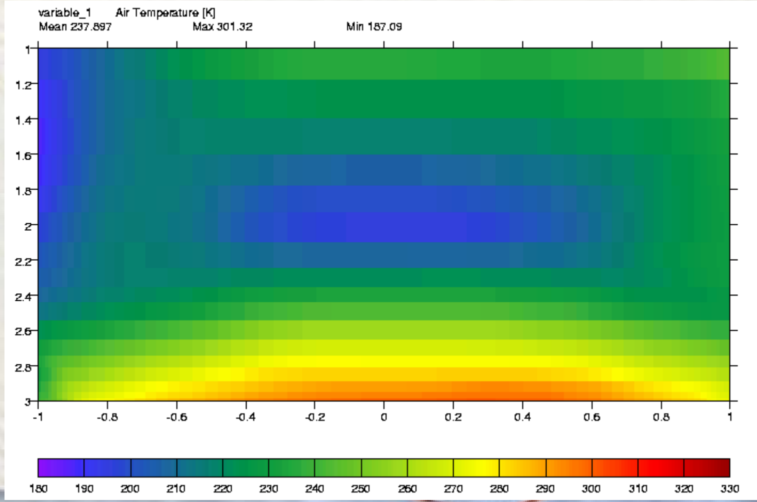
- Note that labels must then be in the units.

- Let’s use another dataset for this case…

2D - Axes Transformation (2)

```
import sys
f=cdms.open(sys.prefix+'/sample_data/ta_ncep_87-6-88-4.nc')
data2=f('ta',time=slice(0,1),squeeze=1)
data2=MV.average(data,-1) # we now have level/lat data
b=x.createboxfill('new')
x.plot(data2,b)
b.xaxisconvert='area_wt' # Area weighted representation of latitudes
b.yaxisconvert='log10' # Log of P representation of Pressure dim
x.plot(data2,b)
```

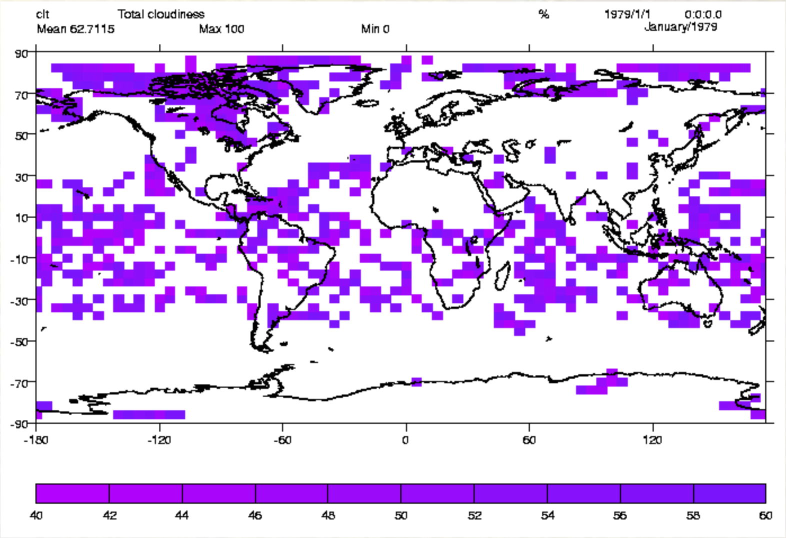




2D - “linear” boxfill_type

- The default boxfill type “linear”, represents grid cells whose values are between the “level1” and “level2” attributes. The color range used to do this are the colors comprised between the “color1” and “color2” attributes, therefore determining how many sub intervals will be used (10 colors meaning 10 segments)

```
b.color1=10
b.color2=19
b.level1=40
b.level2=60
x.plot(data,b)
```



- Note that only grid cell whose values are between 40 and 60 are plotted
- Unfortunately, color 10 to 19 are very similar and therefore a need for a new colormap arises….

2D – Colormaps (1)

- Colormap in VCS consist of 256 colors
- Each colormap has a unique name associated with itself
- VCS is currently limited to **ONE** colormap at a time, but if you clear your plot, you can use mutliple colormap during the same session
- Creating and setting a colormap:

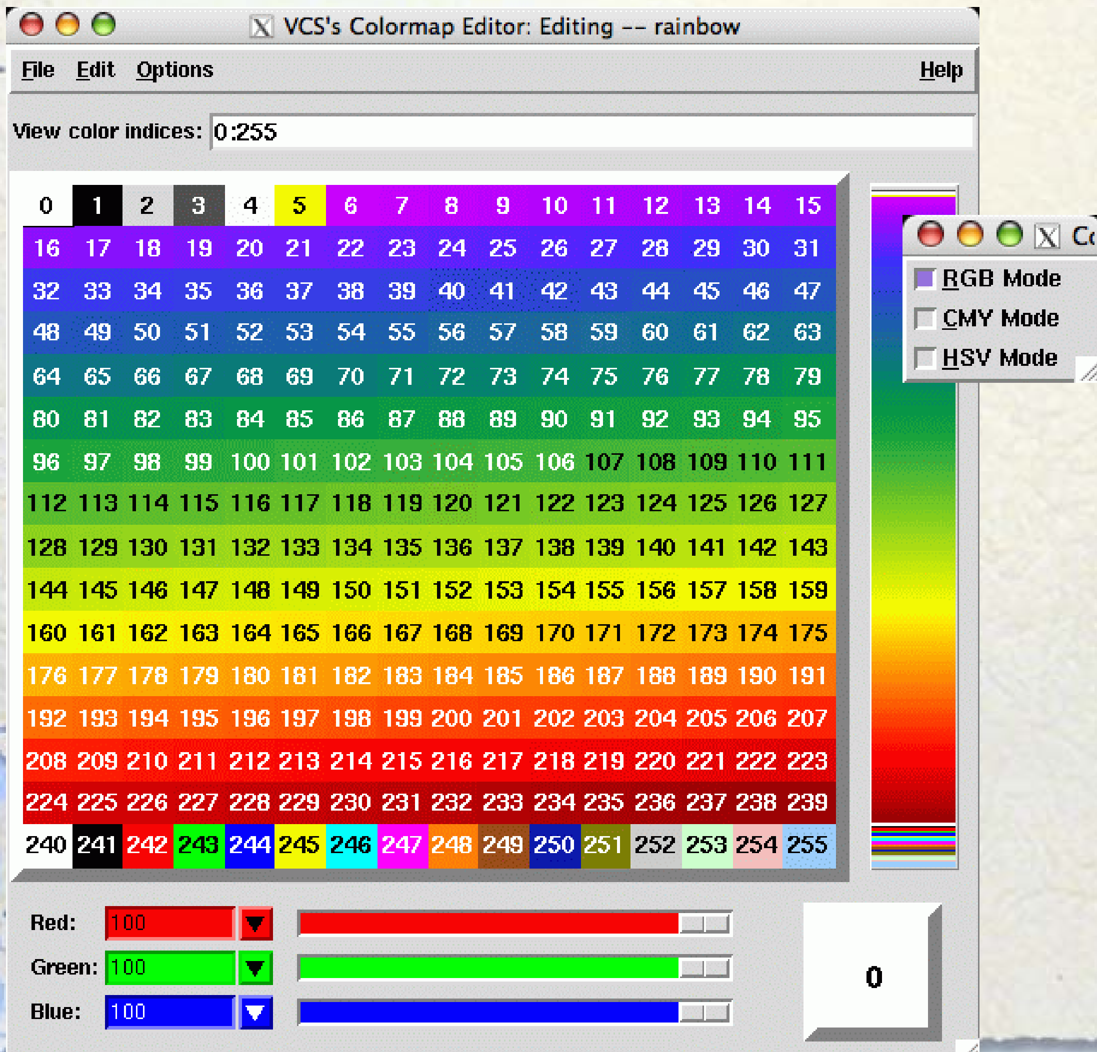
```
x.createcolormap('new')
x.setcolormap('new')
```

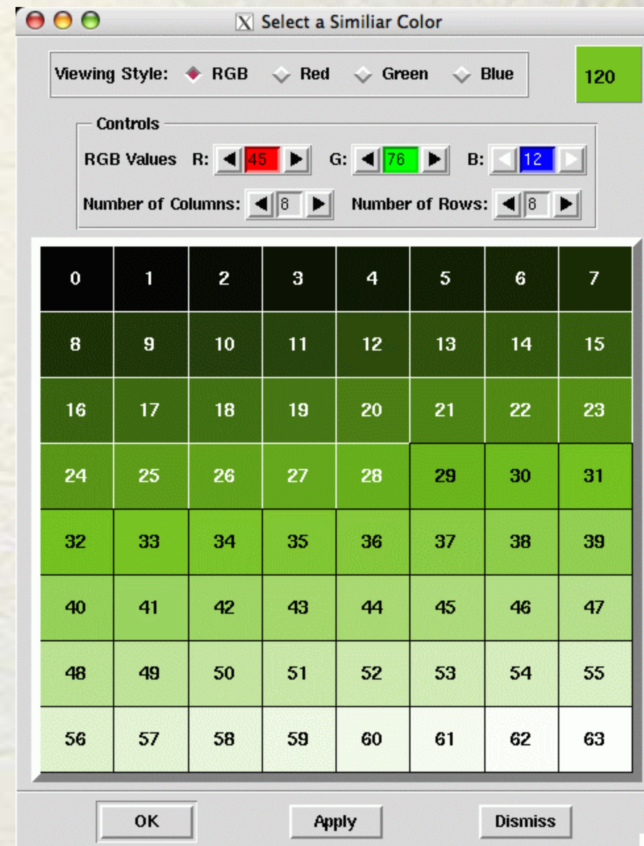
2D – Colormaps (2)

- At this point, since VCS has only 1 colormap you can set/get colorcell values directly from the Canvas object
  Col10=x.getcolorcell(10) # (70,1,99)
- The values returned are RGB values between 0 and 100
- Setting a colorcell is done the same way
  - X.setcolorcell(10,100,0,0) # Sets color 10 to RED
- Note the existence of the colors "sub" module inside genutil
  - r,g,b=genutil.colors.str2rgb('red') # Returns (255,0,0)
  - WARNING: Value of rgb are in between 0 and 255 in this
  - Name=genutil.colors.rgb2str(255,0,0)
- Manipulating colors is much easier from the VCDAT GUI, we strongly recommend to use it to create/edit your colormap, save them to a file or the initial.attributes and then read them in your scripts.

2D - Colormaps from VCDAT

- Colormap GUI let you easily pick/change colors and color model





2D - “log10” boxfill_type

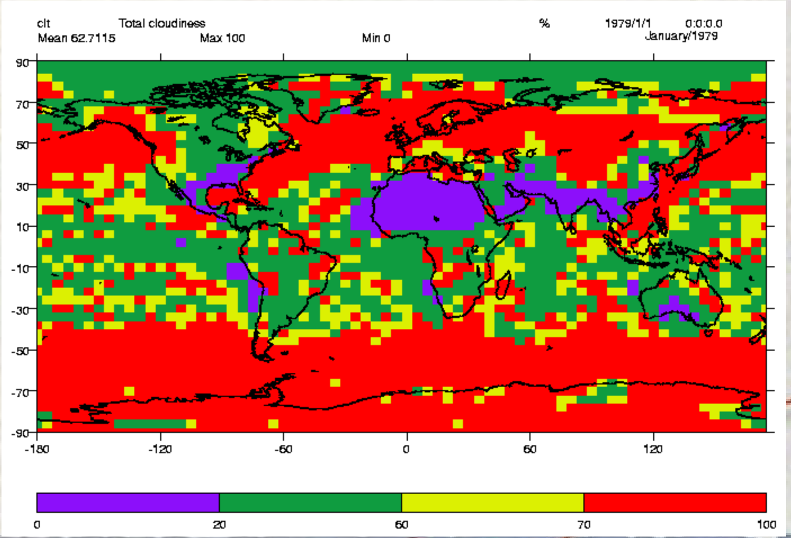
- The boxfill type “log10”, represents grid cells whose values are between the “level1” and “level2” attributes.

- The color range used to do this are the colors comprised between the “color1” and “color2” attributes, therefore determining how many sub intervals will be used (10 colors meaning 10 segments).

- The difference with “linear” is that a log10 is applied on the data before plotting.

2D - “custom” boxfill_type

- The boxfill type “custom”, allows the user to pick the levels interval to use to represent data and the colors associated with each interval.
- Intervals are set via the “levels” attribute:
  - b.boxfill_type=‘custom’
  - b.levels = [0, 25, 50, 75, 100]
- Will represent data into 4 intervals: [0,25], [25,50], [50,75], [75,100]
- Interval length can vary:
  - b.levels = [0, 20, 60, 70, 100]
- Each interval color can be set via the “fillareacolors” attribute
  - b.fillareacolors = [16, 90, 150, 242]



2D - dealing with colors

- Due to the nature of vcs (one colormap) "picking" colors along a colormap can be really painful. Fortunately some vcs built-in functions can help with dealing with levels and colors.

- To automatically pick colors along a range of colors use:
  - vcs.getcolors(levs,colors=range(16,240,split=1,white=240))
  - colors: can be an list/tuple of numbers
  - split: determine if the colors should be split in 2 part, the first part to be used for intervals before values switch from >0 to <0 or <0 to >0, the second part for the other intervals
  - white is used in correlation with split, if an interval goes from <0 to >0 or vice-versa, then this color value is used.

- See doc string for more info.

2D - dealing with levels (1)

- To produce “nlevs” EVEN intervals the first one starting at v1, the last one ending at v2 use:
  - vcs.mkevenlevels(v1,v2,nlevs=10) # Produce “nlevs” EVEN intervals the first one starting at v1, the last one ending at v2
- To produce an automatic “nice” scale for data range going from min to max use:
  - vcs.mkscale(min,max,nc=12,zero=1)
  - Optional Arguments are:
    - nc : Maximum number of intervals (default 12)
    - zero = -1/1/2 (default is 1)

      -1: zero MUST NOT be a contour

      1: zero CAN be a contour (default)

      2: zero MUST be a contour

2D - dealing with levels (2)

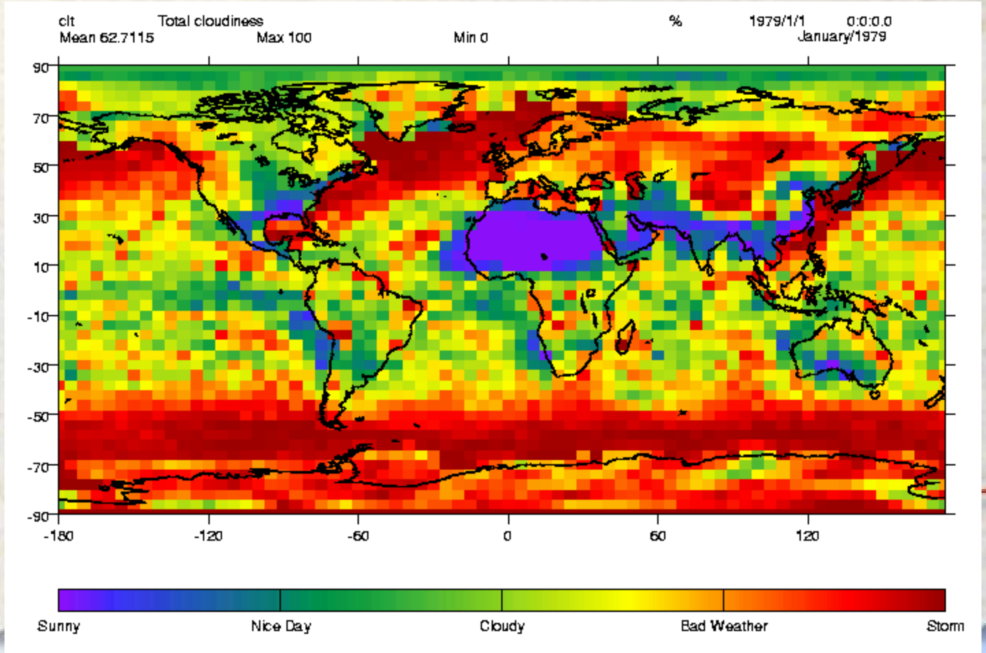
- When dealing with “big” or “small” values it is sometimes nice to use labeled values, e.g 0.0000001 be drawn as: 1.E-7
  - vcs.mklabels([.0000001]) # {9.9999999999999995e-08: '1E-7'}
  - Returns a dictionary of pair : value/”nice”string that can be passed to xticlabels1, etc…

2D - boxfill: legend

- If you chose “linear” or “log10” for the “boxfill_type”, the legend labels can be controlled via the “legend” attribute.
- The “legend” attribute takes a dictionary of value/text pairs

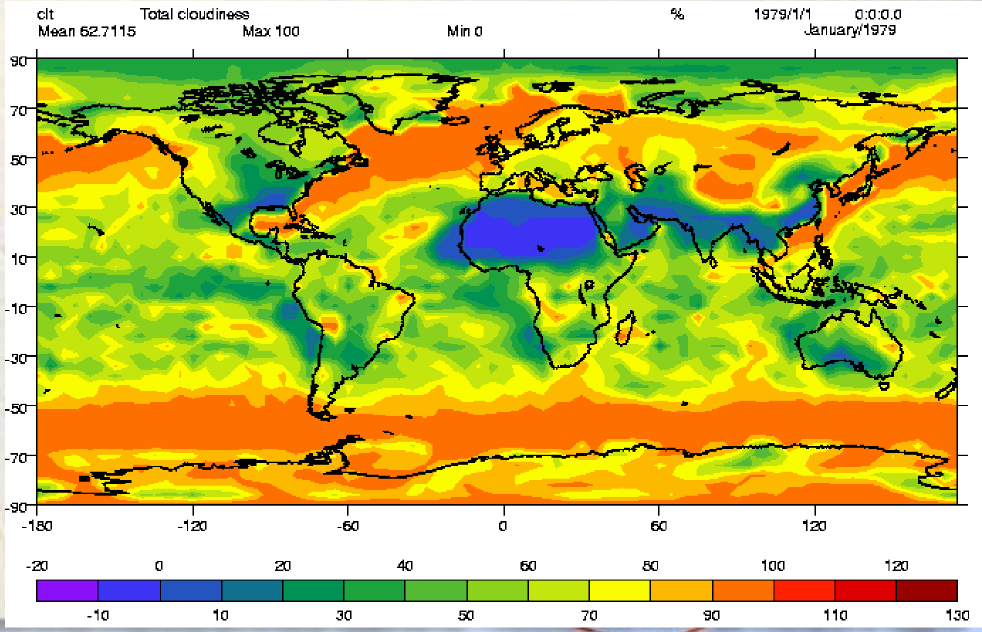
  Leg={0 : “Sunny” , 25: “Nice Day”, 50: “Cloudy” 75: “Bad Weather”, 100: “Storm”}

  b.legend=Leg



2D -“isofill”

- Isofill graphic methods draws filled isocontour
- They are extremely similar to boxfill “custom” type
- Known Limitation:
  - No control on labels position
  - No control on isolines “Smoothness”
- iso=x.createisofill(‘new’)



2D -“isofill”, attributes

- iso.list()

Generic Info
----------Isofill (Gfi) member (attribute) listings ----------
Canvas Mode = 1
graphics method = Gfi # indicates the graphic method type: Graphic Isofill (Gfi)
name = new # Name of the specific graphic method

Projection
projection = linear # projection to use (see projection section)

Labels and Ticks
xticlabels1 = * # 1st set of tic labels, ‘*’ means ‘automatic’
xticlabels2 = * # 2nd set of labels (pos determined by template)
xmtics1 = # 1st set of sub ti for details)
xmtics2 =
yticlabels1 = *
yticlabels2 = *
ymtics1 =
ymtics2 =

World Coordinates
datawc_x1 = 1.00000002004e+20 # world coordinate of 1st x in data area , 1.E20 means auto
datawc_y1 = 1.00000002004e+20
datawc_x2 = 1.00000002004e+20
datawc_y2 = 1.00000002004e+20

Axes transformation
xaxisconvert = linear # Possible conversion of X axis , linear, log, area weighted
yaxisconvert = linear # same for Y

missing = 241 # color to use for missing values
ext_1 = n # draw extension arow before fisr value/segm
ext_2 = n # extension arrow after last num value/segment

Colors
fillareastyle = None # Colors to associate with aach levels section
fillareaindices = None # Colors to associate with aach levels section
fillareacolors = None # Colors to associate with aach levels section

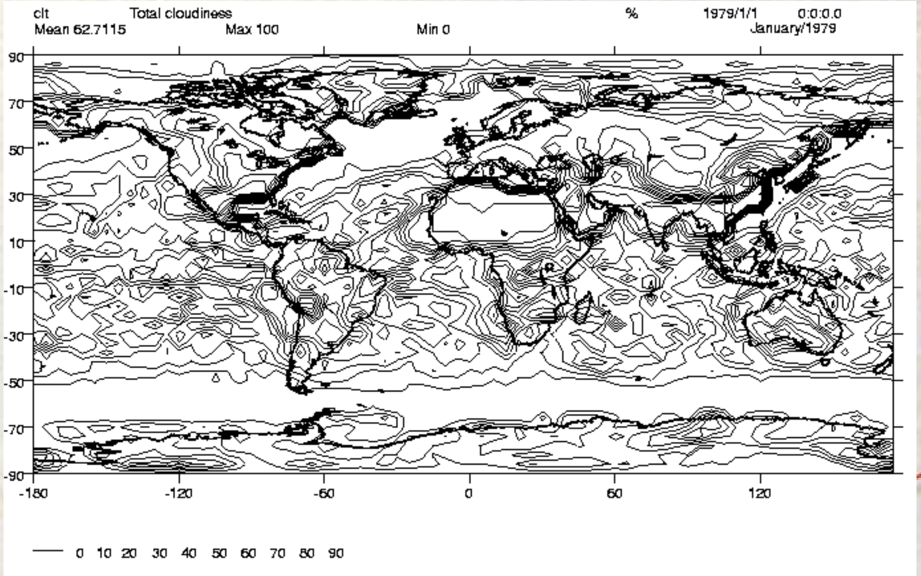
levels = ([1.0000000200408773e+20, 1.0000000200408773e+20],) # Levels of num values to repr in custom mode
legend = None # Dictionary of values/text pair to put on the legend bar

2D -“isofill”, attributes

- As in boxfill “custom” type:
  - “levels” controls the intervals
  - “fillareacolors” controls the colors
  - “ext_1” and “ext_2” control the extension arrows
- As in boxfill “linear” type
  - Legend controls the legend labels

2D - “isoline”

- Isoline, draws isocontours, color, style, can be controlled.
- Limitation:
  - No control on the labels location
  - No control of “smoothness”
- iso=x.createisoline(‘new’)



2D - “isoline”, attributes

- iso.list()

Generic Info	----------Isofill (Gfi) member (attribute) listings ----------
	Canvas Mode = 1
	graphics method = Gfi # indicates the graphic method type: Graphic Isofill (Gfi)
	name = new # Name of the specific graphic method

Projection	projection = linear # projection to use (see projection section)
Labels and Ticks	xticlabels1 = * # 1st set of tic labels, ‘*’ means ‘automatic’ xticlabels2 = * # 2nd set of labels (pos determined by template) xmtics1 = # 1st set of sub ti for details) xmtics2 = yticlabels1 = * yticlabels2 = * ymtics1 = ymtics2 =

World Coordinates	datawc_x1 = 1.00000002004e+20 # world coordinate of 1st x in data area , 1.E20 means auto
	datawc_y1 = 1.00000002004e+20
	datawc_x2 = 1.00000002004e+20
	datawc_y2 = 1.00000002004e+20

Axes transformation	xaxisconvert = linear # Possible conversion of X axis , linear, log, area weighted
	yaxisconvert = linear # same for Y

	label = ‘n’
Isolines definition	line = [‘solid’] Linecolors = [241] Linewidths = [1.0]
Labels definition	text = None textcolors = None
	level = [[0. , 1.e20]]

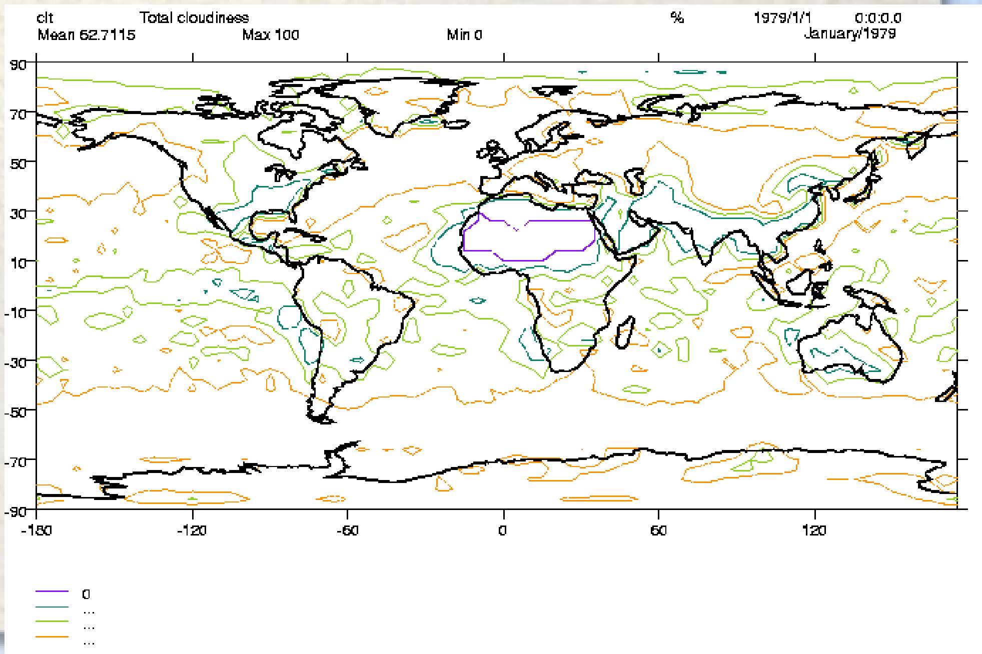
2D - “isoline”, controlling levels

- Isocontours are controlled with the “level” (or “levels”) attribute
  - iso.level= [0 , 25 ,50 ,75, 100]
- Isocontour “aspect” are controlled via:

  iso.line # list of line types “solid”, “dash”, “dot”, “dash-dot”, “long-dash”
  iso.linecolors # list of colors for each isocontour
  Iso.linewidths # list of numbers representing the “thickness” of each contour
- Example

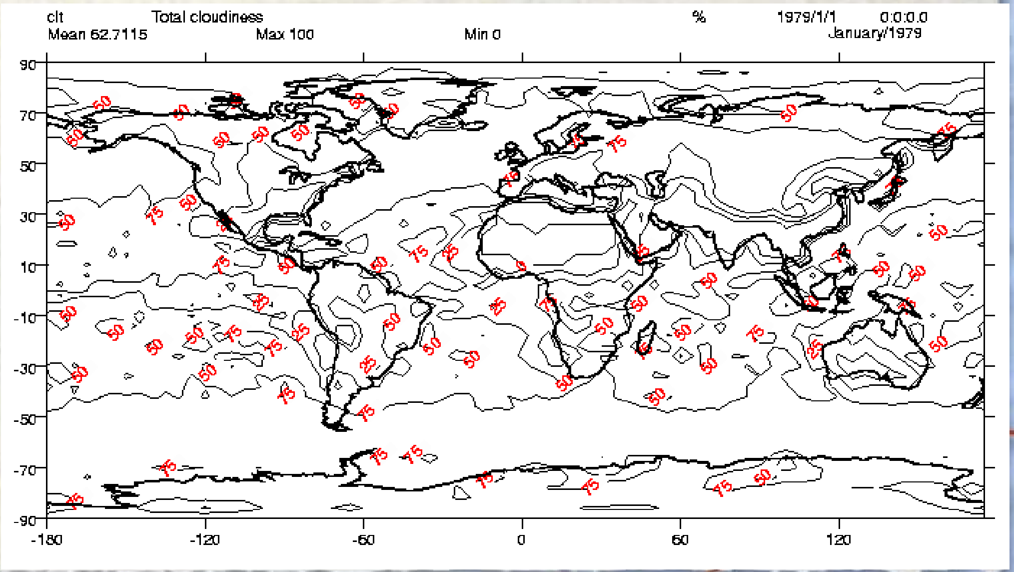
  Levels= [0, 25, 50, 75, 100]
  # vcs.getcolors works on intervals, we need to add one number at the end of levels…
  colors=vcs.getcolors(Levels+[2,])
  iso.levels=Levels
  iso.linecolors=colors
  iso.line = [‘dot’,’dot’,’solid’,’dash’,’dash’]
- Warning:

  Line styles aren’t always converted back to pcm/postscript, it’s a known bug.



2D - “isoline”, controlling labels

- Isocontours labels are controlled via the “label” attribute
- The text attribute allows you to change the font type
- The textcolor attribute allows you to control the color of the labels
- More elaborate control is permitted by passing “VCS text objects” to the text attribute (more later on text objects)
- Example:
  - tt=x.createtext(‘new’)
  - tt=x.label=‘y’
  - tt.angle=-45
  - tt.color=242
  - iso.text=[tt,tt,tt,tt,tt]



2D – “meshfill”

- Meshfill is similar to boxfill “custom” but allows representation of generalized grids, i.e. instead of filling a box, “meshfill” fills cells, of “n” points
- Therefore Meshfill requires an additional array to be passed. This array represents the “mesh”
- This array is generated automatically for Transient Variables recognized by cdms.
- Meshfill allows very creative 2D plots, as long as you know how to generate the “mesh” array. Each cell does not have to have the same number of point.

2D - “meshfill”

- Let’s open an example of “generalized grid” file

```
import sys
f=cdms.open(sys.prefix+'/sample_data/sampleGenGrid3.nc')
data=f('sample') # data shape (2562,)
x.plot(data) # VCS knows automatically to use meshfill
m=x.createmeshfill(‘new’)
```

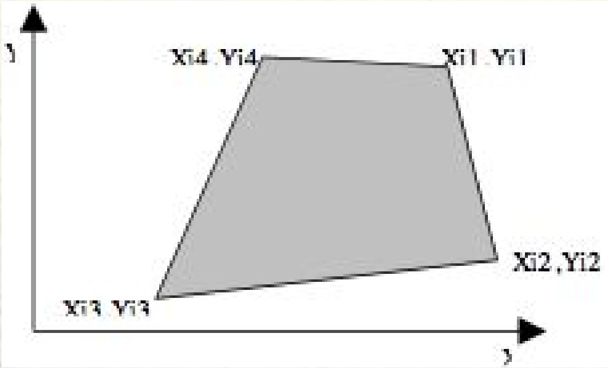
- Getting the “mesh” array
  - Grid=data.getGrid()
  - Mesh = Grid.getMesh()
  - print Mesh.shape # (2562, 2, 6)

2D - “meshfill” - Description of the “Mesh Array”

- The mesh description is an array, its form is: (nelements,2,nvertices) where:
  - ncell represents the number of elements constituting the mesh.
  - 2: represent the Y and X spatial dimension
  - nvertices: represent the number of vertices of each mesh element
- *Note*: if your mesh does not have the same number of vertices per element, then nvertices should be the maximum number of vertices that an element can have, and unused values of vertices should then be set to missing (1.E20 for a Numeric object, or via associated mask for MA and MV)

2D – “meshfill” - Description of the “Mesh Array”

Let’s use for example a mesh constituted of 100 quadrilaterals (or elements).
The mesh array (M) should be dimensioned : (100,2,4)
Now let’s look at how the mesh for the i^{th} quadrilateral (element) (figure 1)



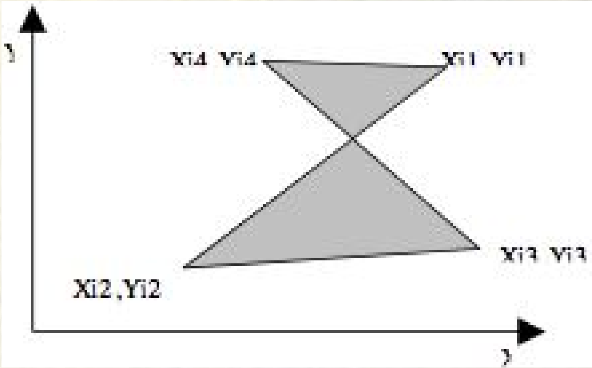


Figure 1 : ith element of data and mesh

Figure 2: misordered mesh elements

For the i^{th} element the the data array (A[i]), the mesh element m=M[i] is 2 dimensional
m[0] has a length of 4 and represents the Y values of the vertices **CONSECUTIVELY**
m[1] has a length of 4 and represents the X values of the vertices **CONSECUTIVELY**

From figure 1: M[i,0]=[Yi1,Yi2,Yi3,Yi4] and M[i,1]=[Xi1,Xi2,Xi3,Xi4]

Note: It is fundamental that the vertices are stored consecutively, the direction does not matter, see figure 2 for an example of bad storage

2D - “meshfill”, attributes

- m.list()

Generic Info
----------Isofill (Gmi) member (attribute) listings ----------
Canvas Mode = 1
graphics method = Gfm # indicates the graphic method type: Graphic Meshfill (Gfm)
name = new # Name of the specific graphic method

Projection
projection = linear # projection to use (see projection section)

Labels and Ticks
xticlabels1 = * # 1st set of tic labels, ‘*’ means ‘automatic’
xticlabels2 = * # 2nd set of labels (pos determined by template)
xmtics1 = # 1st set of sub ti for details)
xmtics2 =
yticlabels1 = *
yticlabels2 = *
ymtics1 =
ymtics2 =

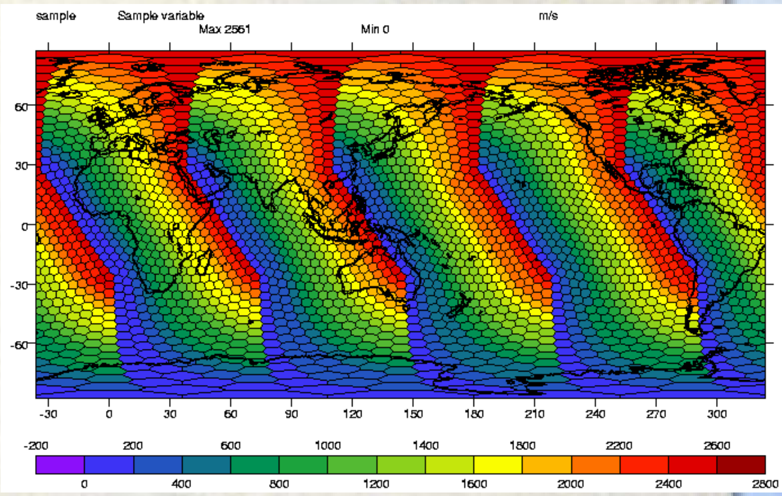
World Coordinates
datawc_x1 = 1.00000002004e+20 # world coordinate of 1st x in data area , 1.E20 means auto
datawc_y1 = 1.00000002004e+20
datawc_x2 = 1.00000002004e+20
datawc_y2 = 1.00000002004e+20

Axes transformation
xaxisconvert = linear # Possible conversion of X axis , linear, log, area weighted
yaxisconvert = linear # same for Y

Levels & colors
levels = ([1.0000000200408773e+20, 1.0000000200408773e+20],) # Levels of num values to repr in custom mode
fillareacolors = None # Colors to associate with aach levels section
fillareastyle = None # Colors to associate with aach levels section
fillareaindices = None # Colors to associate with aach levels section
legend = None # Dictionary of values/text pair to put on the legend bar
missing = 241 # color to use for missing values
ext_1 = n # draw extension arow before fisr value/segm
ext_2 = n # extension arrow after last num value/segment
mesh = 0
wrap = [0.0, 0.0]

2D - “meshfill”, attributes

- Levels/ Colors, etc… are controlled the same way than isofill
- New attributes are:
  - mesh: Draws the mesh if set to 1

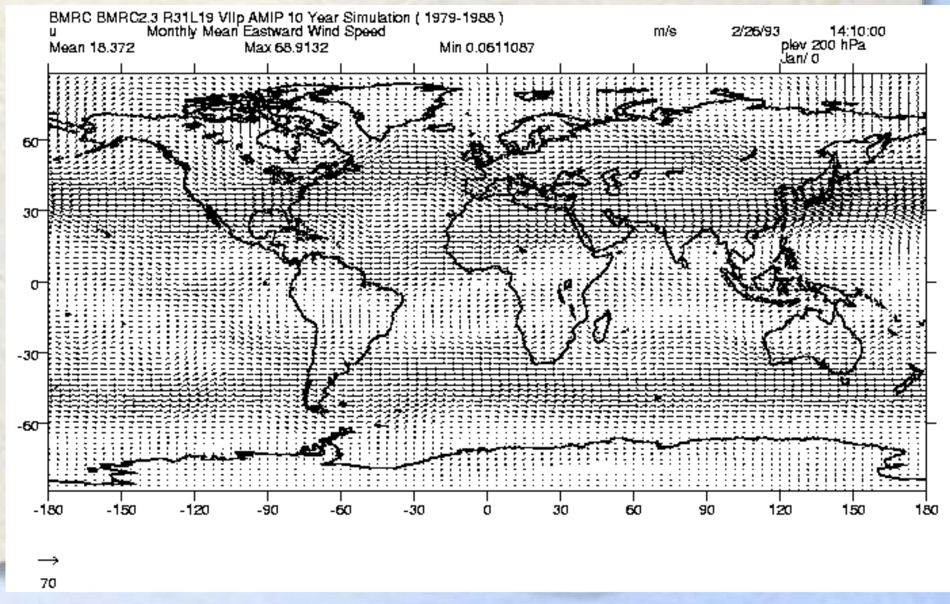


  - wrap: Specifies the wrapping value in each x/y dimension

2D - “vector”

- The “Vector” graphic method represents the combination of 2 arrays, via “vector” the first array representing the “X” axis component and the second array representing the “Y” axis component.
- In the same “clt.nc” sample file we can retrieve 2 fields “u” and “v” and use them to draw wind fields.

```
f=cdms.open(sys.prefix+’/sample_data/clt.bc’)
u=f(‘u’)
v=f(‘v’)
vec=x.createvector(‘new’)
x.plot(u,v,vec)
```



2D - “vector”, attributes

- vec.list()

	----------Isofill (vi) member (attribute) listings ----------

Generic Info	Canvas Mode = 1
	graphics method = Gv # indicates the graphic method type: Graphic Vector (vi)
	name = new # Name of the specific graphic method

Projection	projection = linear # projection to use (see projection section)
Labels and Ticks	xticlabels1 = * # 1st set of tic labels, ‘*’ means ‘automatic’ xticlabels2 = * # 2nd set of labels (pos determined by template) xmtics1 = # 1st set of sub ti for details) xmtics2 = yticlabels1 = * yticlabels2 = * ymtics1 = ymtics2 =
World Coordinates	datawc_x1 = 1.00000002004e+20 # world coordinate of 1st x in data area , 1.E20 means auto datawc_y1 = 1.00000002004e+20 datawc_x2 = 1.00000002004e+20 datawc_y2 = 1.00000002004e+20
Axes transformation	xaxisconvert = linear # Possible conversion of X axis , linear, log, area weighted yaxisconvert = linear # same for Y
Vectors definition	line = Noe linecolor = None linewidth = None
	scale = 1.0 alignement = center type= arrows reference = 1.E20

2D - “vector”, attributes

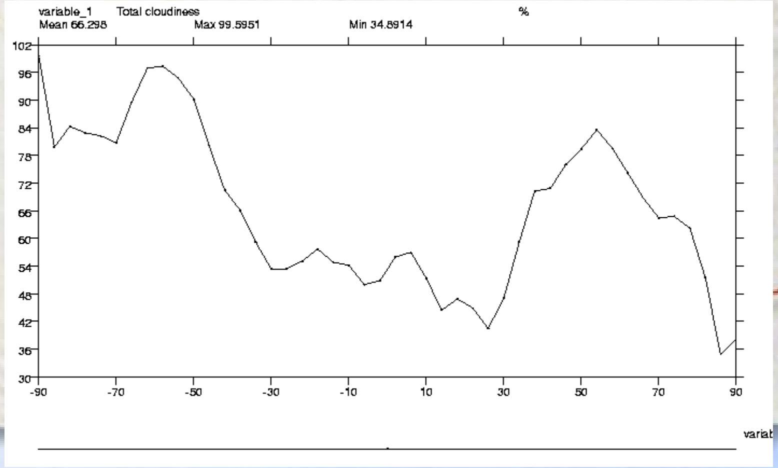
- The vectors line/color/width are defined the same way isolines are, except the attribute takes only 1 argument not a list.

- Additional definition are:
  - reference : Numerical value associated with the “standard” arrow, if set to 1.E20, then it will be the length of the longest arrow
  - scale: factor to “scale” the “reference” arrow
  - type: type of the arrows: “arrows”, “barbs” or “solidarrows”
  - alignment: how to align the “arrows” relative to the center of the cell “head”, “center”, “tail”

1D - “VCS”

- All 1D plots in VCS basically work the same way. There are 4 types of 1D graphic method, we’ll start with the basic: Yxvsx, which stands for Y(x) vs x
- This graphic method draws a 1D array (Y) as a function of its 1D axis (x)
- Example zonal mean of the first time point of our data array

```
zm=MV.average(data[0],1) # Zm.shape is (46,)
x.plot(zm) # automatically knows to plot 1D with yxvsx
yx=x.createyxvsx(‘new’)
x.plot(zm,yx) # same
```



1D - “yxvsx”, attributes

- yx.list()

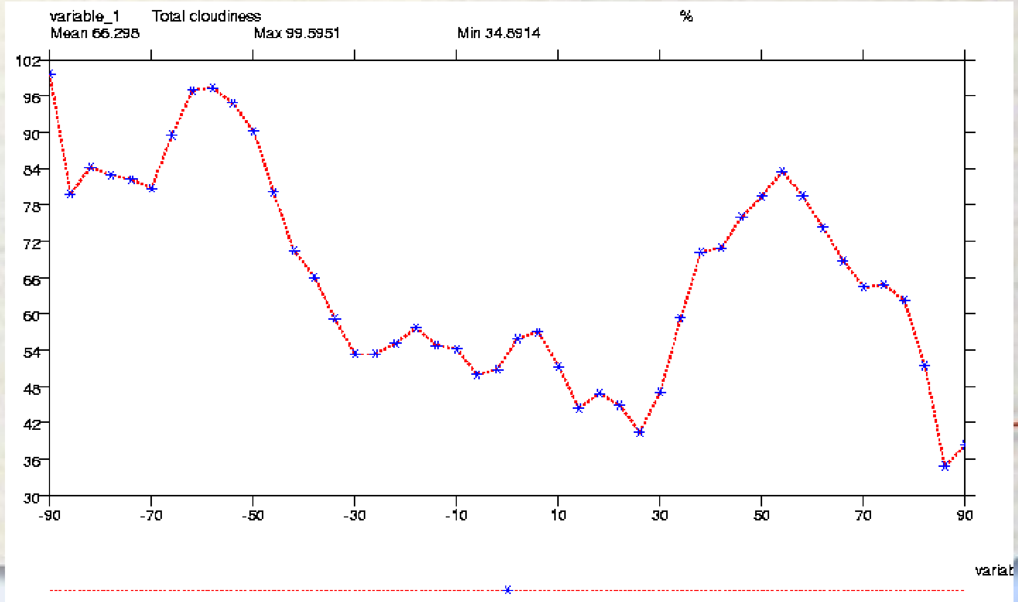
Generic Info	----------Isofill (Gfi) member (attribute) listings ----------
	Canvas Mode = 1
	graphics method = Gfi # indicates the graphic method type: Graphic Isofill (Gfi)
	name = new # Name of the specific graphic method

Projection	projection = linear # projection to use (see projection section)
Labels and Ticks	xticlabels1 = * # 1st set of tic labels, ‘*’ means ‘automatic’ xticlabels2 = * # 2nd set of labels (pos determined by template) xmtics1 = # 1st set of sub ti for details) xmtics2 = yticlabels1 = * yticlabels2 = * ymtics1 = ymtics2 =
World Coordinates	datawc_x1 = 1.00000002004e+20 # world coordinate of 1st x in data area , 1.E20 means auto datawc_y1 = 1.00000002004e+20 datawc_x2 = 1.00000002004e+20 datawc_y2 = 1.00000002004e+20
Axis transformation	xaxisconvert = linear # Possible conversion of X axis , linear, log, area weighted
Line definition	line = None linecolor = None linewidth = None
Markers definition	marker = None markercolor = None markersize = None

1D - “VCS” Yxvsx attributes

- As in isoline, or vector, line, linecolor, linewidth, determine the line
- marker, markercolor, markersize, determine the markers to be drawn

```
yx.line='dot'
yx.linecolor=242
yx.linewidth=2
yx.marker='star’ # use None for no marker
yx.markercolor=244
yx.markersize=5.
```



1D - “VCS” Xyvsx, Xvsy, scatter

- Other 1D graphic method, work very similarly
- Xyvsy does the same thing except the X and Y axes are flipped relatively to a Yxvsx.
- XvsY is the same thing as YxvsX except it takes 2 data array (X and Y) as arguments, therefore the values on the horizontal axis are not taken from the “axis” definiton of the Y array but from the values of a first (X) array
- To reproduce the Yxvsx plot we would do:
  - ax=zm.getAxis(0)[:]
  - xy=x.createxvsy(‘new’)
  - x.plot(ax,zm,xy)
- Note that this allows for distortion on both X and Y direction (xaxisconvert and yaxisconvert)
- Scatter basically works as XvsY except both X and Y MUST have the same Axis!
- In the previous example we would need to add the following line:
  - ax=MV.array(ax)
  - ax.setAxis(0,zm.getAxis(0))
  - sc=x,createscatter(‘new’)
  - x.plot(ax,zm,sc)

VCS Templates - Generalities

- Now that we’ve seen how to display data, it is important to understand the “templates” concept.
- Template are used to tell VCS “where” to draw objects on the canvas (whereas Graphic Method would tell “how” to draw them”)
- There’s basically 4 different aspect of the plot that are controlled by the templates
  - Text location for things, like title, comments, name, etc..
  - Data area
  - Tick marks and labels location
  - Legend
- Each element of the template (e.g “data”) can be turned on/off or moved on top/below other elements on the page via its “priority” attribute
- Template object are created via the “createtemplate” command

  t=x.createtemplate(‘new’)
- All elements of a template object can be listed via the list() function

  t.list()
- Alternatively, a single elements’s attributes can be listed, e.g.:

  t.data.list()

VCS Templates - Text elements and their sources (1)

- Some text can be automatically displayed on the Canvas, the values are taken from the slab itself, here is a mapping of template attribute names and slab names:

  title: slab long_name attribute
  dataname: slab id attribute
  source: slab’s source attribute
  file: slab’s filename attribute
  logical_mask : ???
  function : ???
  transformation: ???
  units: slab’s units attribute
  crdate: If slab has time dimension, then date of the slice plotted
  crtime: If slab has time dimension, then time of slice plotted
  comment1/2/3/4: slab’s attribute comment1/2/3/4

VCS Templates
Text elements and their sources (2)

- We need to remind here that the x/y/z/t dimension in VCS correspond to the slab's -1/-2/-3/-4 dimensions!
  - x/y/z/t/name: name of the -1/-2/-3/-4 dimension
  - x/y/z/t/units: units for the -1/-2/-3/-4 dimension
  - x/y/z/t/value: value for the -1/-2/-3/-4 dimension

    (only for dim not plotted as data, e.g. plotting a time/lat/lon slab with boxfill then tvalue will represent the time plotted, this will change for each animation frame)

  - min/max/mean: min/max/mean of the data plotted

    WARNING: These values are for the entire slab passed, e.g datawc restriction won't be taken into account here!

VCS Templates
Definition of text elements

- All the elements described before behave the same, let's look at the dataname element:

```
t.dataname.list()
member =  dataname
    priority = 1
    x = 0.0500000007451
    y = 0.922999978065
    texttable = default
    textorientation = default
```

priority: 0 means off, anything else represent the layer on which to draw the element, the higher value being drawn on top of lower values

x/y: x/y location of the element in % of the page !

texttable : vcs texttable element to use (see next slide)

textorientation: vcs textorientation element to use (see next slide)

VCS Templates
Texttable and textorientation (1)

- Textobject in VCS have 2 component a texttable component and a textorientation element:
  - Tt=x.createtexttable('new')
  - Tt.list()

```
----------Text Table (Tt) member (attribute) listings ----------
Canvas Mode = 1
secondary method = Tt
name = new
font = 1
spacing = 2
expansion = 100
color = 1
```

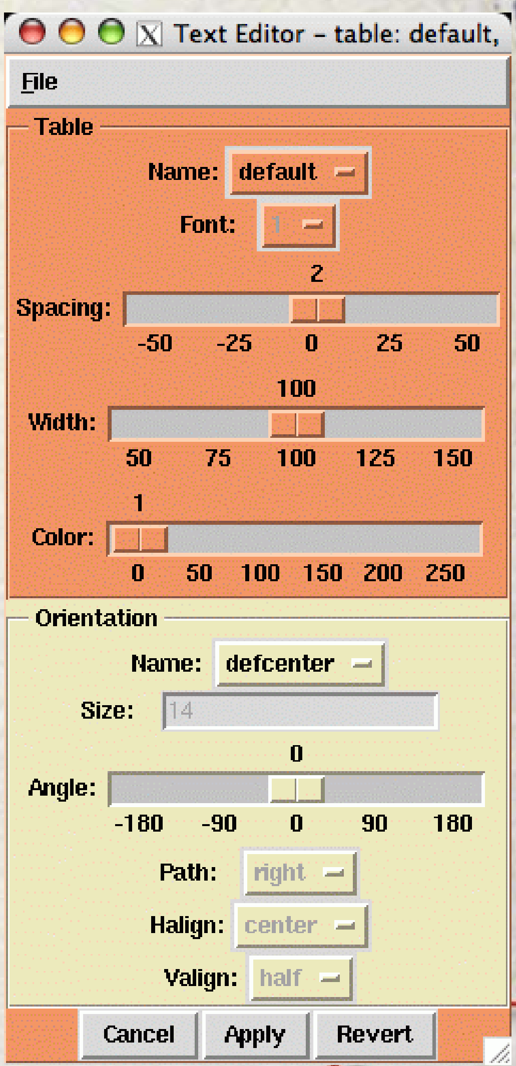
VCS Templates
Texttable and textorientation (2)

- To=x.createtextorientation('new')
- To.list()

```
----------Text Orientation (To) member (attribute) listings ----------
Canvas Mode = 1
secondary method = To
name = new
height = 14
angle = 0
path = right   # right left,up,down
halign = left # left,center,right
valign = half #top,cap,half,base,bottom
```

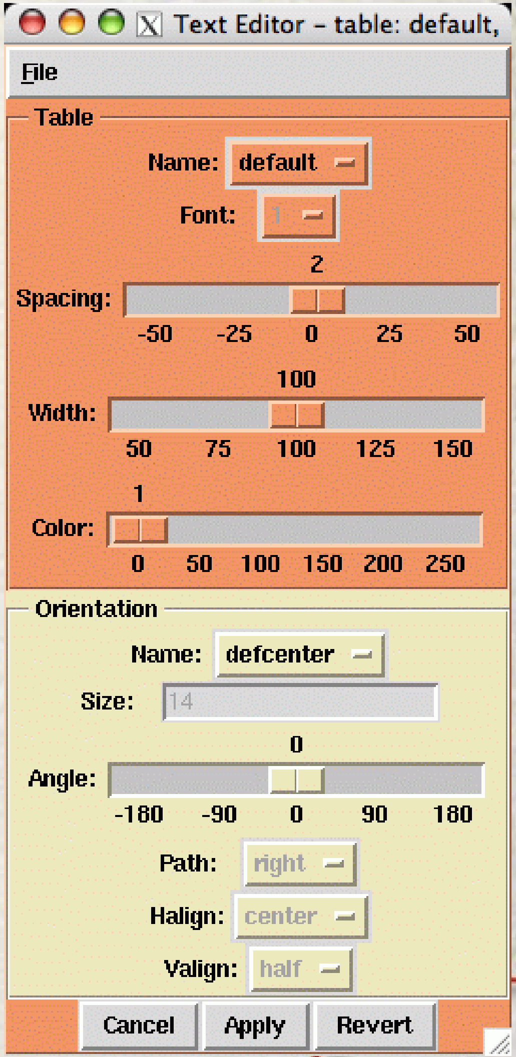
VCS Templates
Controlling ticks and labels (1)

- Tics and labels work similarly on the x and y axis, we'll only describe the X axis ones
- T.xlabel1/2 # control the location/aspect of the labels1/2
  - priority
  - Y # location of label in % of page
  - texttable
  - textorientation
- T.xtic1/2 # control the location/aspect of the labels1/2
  - priority
  - y1 # start of tic mark in % of page
  - y2 # end of tic mark in % of page
  - line # vcs line object name (see later)



VCS Templates
Controlling ticks and labels (2)

- T.xmintic1/2 # control the location/aspect of the labels1/2
  - priority
  - y1 # start of tic mark in % of page
  - y2 # end of tic mark in % of page
  - line # vcs line object name (see later)
- Note
  - Xtic1 and xlabel1 values are controlled via the graphic method: xticlabels1
  - their “X” locations are determined via the “data” eleement and domain used

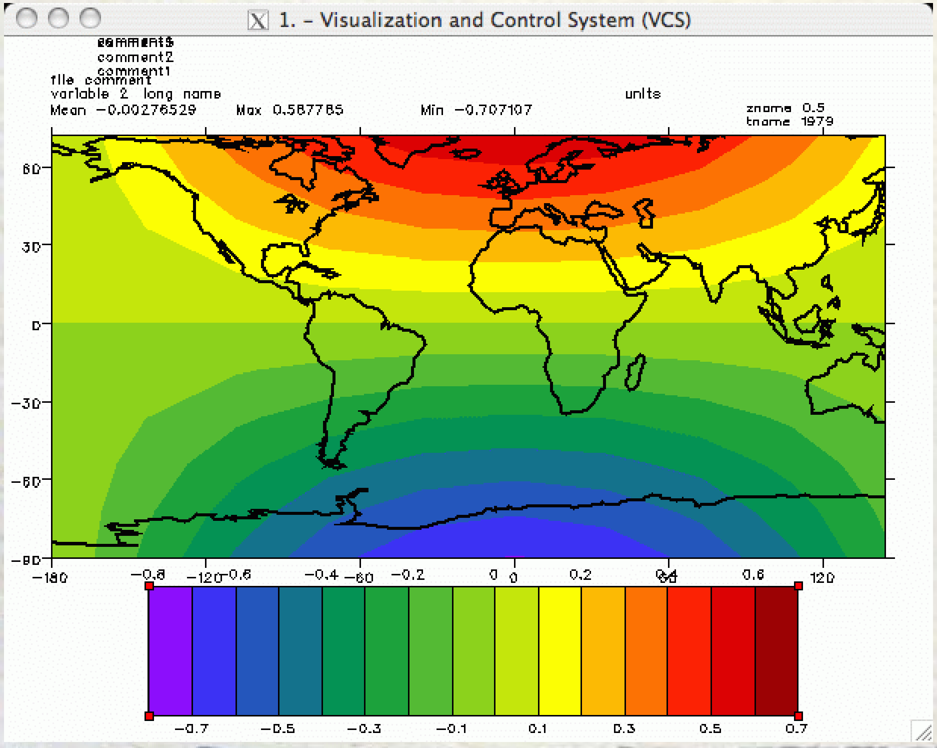


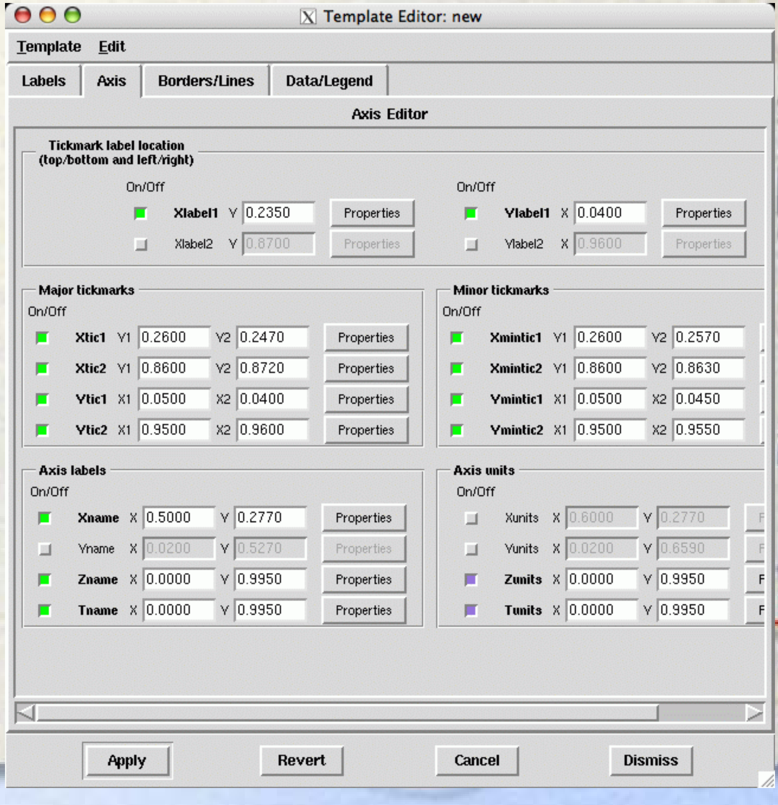
VCS Templates
Controlling legend and data area

- These are controlled via the “data” and “legend” elements
- The data element simply consist of
  - priority
  - x1,x2,y1,y2
- Note: data.x1 corresponds to graphicmethod.datawc_x1
- In addition the “legend” element also has the “line”, “texttable” and “textorientation” attributes
- Finally the data area is surrounded by the “box1” element, with the following attributes: priority,x1,x2,y1,y2,line
- Note that template object have 4 box and 4 line elements, but beside box1, these are rather obsolete and primitive objects should be used instead

VCS Templates
Simplifying your life! (1)

- Controlling all the aspect of a template can be really tedious, it is recommended to set up your template once, save it and reuse it for ever.
- In order to edit your template once and for all we recommend to use the VCDAT template editor!





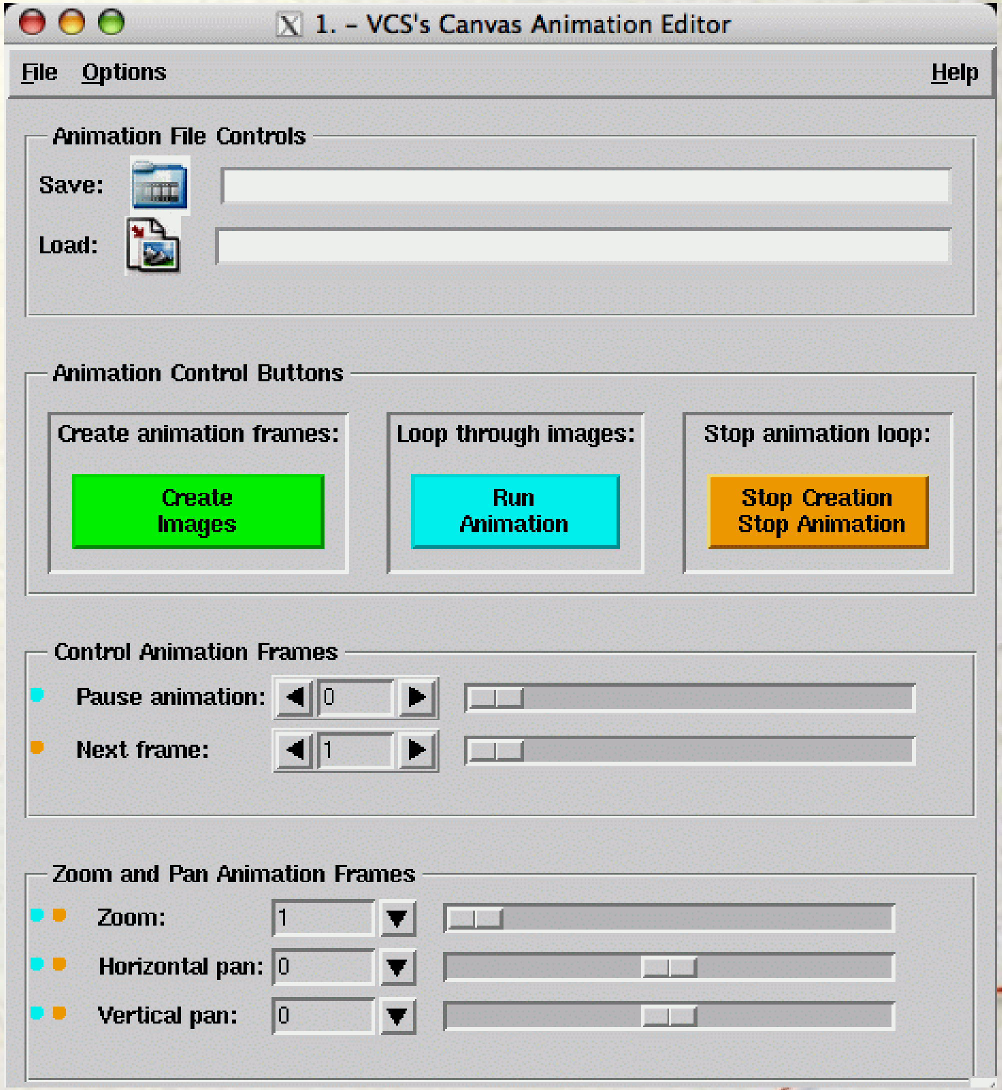
VCS Templates
Simplifying your life! (2)

- Command line manipulation of templates is also made easier with the following command which move all elements at once!
  - T.move(percent, axis)
  - T.moveto(x,y) # move the data.x1,data.x2 corner to x/y
  - T.scale(scale,axis='xy',font=-1) # scale the template along x,y,xy can also scale fonts (automatic for xy direction only)

VCS
Extra dimensions/Animation

- Although graphic method draw only 2D or 1D plots we've seen that we can pass arrays with extra dimension, these dimensions are then used for the animation.

- Animation are best run via the VCDAT GUI but can be controlled from the command line.



“Secondary” VCS Objects

- Secondary VCS object consist of
  - Fillarea : polygons
  - Lines
  - Text
  - Marker

“Secondary” VCS Object: fillarea (1)

----------Fillarea (Tf) member (attribute) listings ----------

Canvas Mode = 0

secondary method = Tf

name = new

style = ['solid']

index = [1]

color = [241]

priority = 1

viewport = [0, 1, 0, 1]

worldcoordinate = [0, 1, 0, 1]

x = None

y = None

projection = default

- Multiple fillarea polygon can be passed via multiple list of x/y, color/ etc… Last element repeated to match the number of list passed into x/y

“Secondary” VCS Object: fillarea (2)

- The only fundamental attributes to understand in the secondary objects are:

  viewport vs **worldcoordinate**

- The **viewport** values (x1,x2,y1,y2) correspond to area of the canvas on which this primitive can be drawn, everything outside of the viewport will be masked.

- The **worldcoordinate** (wcx1,wcx2,wcy1,wcy2) correspond to the World Coordinate values at each angle of the viewport, x/y values of the object are expressed in “worldcoordinate”, by default there is no distinction since they both go 0,1,0,1.

“Secondary” VCS Object: fillarea (3)

- Here is an example of where the difference can be useful: let’s admit you let the user chose between 3 different type of template, and 3 different “Zooms” (magnifications) of an area. And you want to place a marker/text/rectangle/line at a specific location (station data for example)

- You’ll have to set the viewport to the same values as template.data.x1, x2, y1,y2 values and the worldcoordinate to the same values as your graphicmethod.datawc_x1,x2,y1,y2.

- Then your primitive object x/y coordinate will be in real word coordinates, and will always be at the right location no matter which template or graphic method the user picked.

“Secondary” VCS Object: text(combined)

----------Text combined (Tc) member (attribute) listings ----------
Canvas Mode = 0
secondary method = Tc
----------Text Table (Tt) member (attribute) listings ----------
Tt_name = new
font = 1
spacing = 2
expansion = 100
color = 1
priority = 1
string = None
viewport = [0, 1, 0, 1]
worldcoordinate = [0, 1, 0, 1]
x = None
y = None
projection = default
----------Text Orientation (To) member (attribute) listings ----------
To_name = new
height = 14
angle = 0
path = right
halign = left
valign = half

- Passing list to x/y allows multiple text with same attributes

“Secondary” VCS Object: line

----------Line (Tl) member (attribute) listings ----------
Canvas Mode = 0
secondary method = Tl
name = new
type = ['solid']
width = [1.0]
color = [241]
priority = 1
viewport = [0, 1, 0, 1]
worldcoordinate = [0, 1, 0, 1]
x = None
y = None
projection = default

“Secondary” VCS Object: marker

----------Marker (Tm) member (attribute) listings ----------

Canvas Mode = 0

secondary method = Tm

name = new

type = ['dot']

size = [1.0]

color = [241]

priority = 1

viewport = [0, 1, 0, 1]

worldcoordinate = [0, 1, 0, 1]

x = None

y = None

projection = default

1D - “xmgrace”

- User familiar with the “xmgrace” package will be happy to know that they can now drive it from CDAT.
- X=genutil.xmgrace.init()
- X.plot(data) # X values taken from data axis
- See tutorial for more info